Treewidth: Computations, Approximations, and Applications

Iniyaal Kannan
iniyaalkannan@berkeley.edu

Jonathan Tay
jontay999@berkeley.edu

Calvin Yan
calvinyan@berkeley.edu

***Abstract*—This survey aims to provide a comprehensive review of the topic of treewidth; its meaning, significance and application towards other problems in computer science, and computation, in a manner that is accessible to undergraduate students. We place a particular emphasis on parameterized algorithms for computing tree decomposition, both exact and approximate, and discuss our implementation of a seminal result within this class of algorithms.**

1. Introduction

Treewidth is a crucial graph parameter, that has a significant role in domains like artificial intelligence, constraint-satisfaction and logical-circuit design [5]. It is an especially interesting metric because most problems that are infeasible to compute on general graphs are solvable in linear time if a tree decomposition of bounded width is provided as input instead of the original unmodified graph. Given a graph G and an integer k as input, (1) determining whether the treewidth of G is at most k and (2) finding a *tree decomposition* of width at most k are known to be NP-complete problems. It is an open question if a constant-factor treewidth approximation can be found in linear time [4]. There have been several approaches proposed for computing treewidth. Algorithms proposed by Arnborg et al. [], Bodlaender [9], Fomin et al. [0] produce exact solutions to the treewidth algorithm in exponential time. There are several other important approximation results as well. Bodlaender et al. gave a 5-approximation algorithm with runtime $n \times 2^{O(k)}$ (n is the number of vertices in the vertex set and k is the specified treewidth) [10] Bodlaender et al. also gave a 3-approximation algorithm that runs in $n \log n \times 2^{O(k)}$ [7]

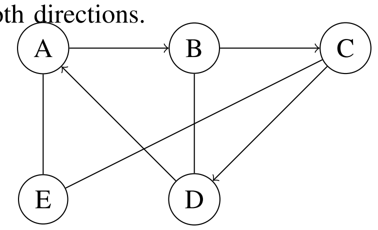
In this project, we implement Tuuka Korhonen's 2-approximation algorithm for treewidth [1]. Their algorithm improves upon Bodlaender et al's paper and produces a 2-approximation in $2^{O(k)}n$ time.

2. Background

2.1. Graphs

A graph is a mathematical object that consists of a set of vertices and a set of edges that connect pairs of vertices. It is represented as $G(V, E)$. V represents the vertex set or number of nodes in the graph and E represents the edge set. Optionally it could also include a cost associated with traversing the edge.

The edges that represent graphs can be directed or undirected. A directed edge, $e \in E$ from node X to Y means e can only be traversed in the X to Y direction whereas an undirected edge means e can be traversed from both directions.



Graphs are used to capture the relationships between the vertices that represent the graph. Graph computations like traversals and clusterings are extremely important to derive insights about graph structures. There are several graph computations of practical significance that are NP-complete [5]

- *Hamiltonian cycle problem:* Given an undirected graph, is there a cycle that visits every vertex exactly once?
- *Traveling salesman problem:* Given a complete weighted graph and a positive integer k, what is the shortest possible tour that visits every vertex exactly once and has a total weight at most k?
- *Vertex cover problem:* Given an undirected graph and a positive integer k, is there a set of at most k vertices that touches every edge in the graph?
- *Clique problem:* Given an undirected graph and a positive integer k, is there a subset of k or more vertices that is a complete graph i.e. all vertices are connected to each other?
- *Independent set problem:* Given an undirected graph and a positive integer k, is there a set of at least k vertices that are not connected to each other?

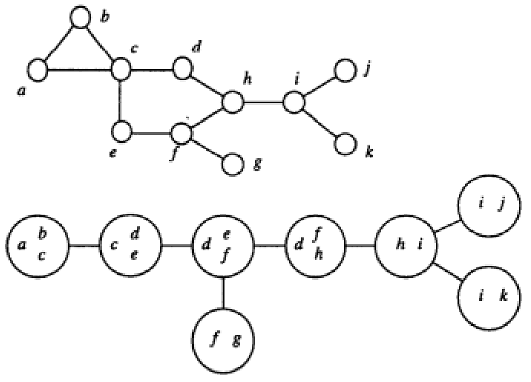
2.2. Treewidth

Definition: A *tree decomposition* (B, T) of a graph $G = (V, E)$ is a tree $T = (V', E')$ and set $B = \{B_1, B_2, \ldots, B_{|V'|}\}$ that obeys the following properties:

- Each vertex $i \in V'$ corresponds to a *bag* of vertices $B_i \subseteq V$.
- $V = \bigcup_{i\in V'} B_i$. Every vertex in G appears in some bag in T, and the bags contain no vertices outside of V.
- For any $(u, v) \in E$ there exists some B_i such that $\{u, v\} \subseteq B_i$.
- For any $v \in V$, the set $\{i \in V' : v \in B_i\}$ is a subtree of T.

Then the *width* of a tree decomposition is equal to $\max_{i\in V'} |B_i| - 1$.

We will illustrate tree decomposition using the following example with width 2 [8]:



Definition: The *treewidth* of a graph G is the minimal width of all possible tree decompositions of G.

The treewidth provides a measure of how "tree-like" the graph is. Trees are algorithmically easier to deal with and the complex constraints that make treewidth an NP-Complete problem are more relaxed with bounded treewidth. Some NP-Hard problems with bounded treewidth can be solved efficiently using dynamic programming. Classes of important graphs have bounded treewidth [**?**]:

- Trees and forests: bounded treewidth 1
- Series parallel and outerplanar graphs: bounded treewidth 2
- Halin graphs: bounded treewidth 3
- Almost trees with k additional edges: bounded treewidth k

This method can be used to brute force solve the treewidth problem exactly:

(1) Enumerate all possible tree decompositions of the graph

(2) For each tree decomposition, compute its treewidth: Take the maximum size of the bags minus 1 in the decomposition. The minus one is a normalizing factor when computing treewidth. This is done to ensure the treewidth of a tree is equal to 1.

(3) The smallest treewidth over all computed treewidths is the treewidth of the graph.

2.3. NP-Completeness of Treewidth

Theorem 2.3.1 (Courcelle's Theorem). *Let F be a formula in extended monadic second order logic. On graphs with constant treewidth, F can be decided in polynomial time.*

These NP-hard problems can be decided in polynomial time on a graph with bounded treewidth [cite]

- Deciding SAT, where G is the primal graph of the SAT instance
- Deciding colorability of G
- Searching for a Hamiltonian cycle in G
- Travelling Salesperson Problem in G
- Finding largest clique in G
- Finding largest vector disjoint path in G

We will give a high-level explanation of algorithms for solving graph colorability as well as maximising CSP in linear times if we are given a suitable tree decomposition of the graph as an input.

2.3.1. Determining colorability on a nice tree decomposition.

A *nice* tree decomposition is a binary tree with the following types of nodes [2]:

- Leaf Nodes: have no children and a bag size of 1
- Introduce Nodes: have one child; the child has the same vertices as parent with one deleted
- Forget Nodes: have one child; the child has the same vertices as parent with one added
- Join Nodes: have two children; both are identical to the parent

Dynamic programming [2] can be used to solve the colorability of such a graph. There exists an algorithm to convert a tree decomposition into a nice decomposition.

The boolean variable $E[X, c]$ is true *iff* c can be extended to a valid coloring on X and the children of X (where X is a node and c is a coloring).

We want to know if $E[\text{root}, c]$ is True. So, we use a top-down Dynamic Programming approach by checking $E[\text{node}, c]$ recursively at every level to determine if a valid coloring from the root can be achieved.

- If X is a leaf: Any coloring is valid; $E[X, c]$ is true for all c.
- If X introduced vertex u: For each coloring c, the two conditions it must meet are (1)it is a valid coloring on X and (2) it is valid among its children nodes.
- If X forgot vertex u: Need to check if c can be extended to u. $E[X, u]$ is True *iff* there exists a color c_1 extending c to $X + u$.
- If X joined the nodes Y and Z: The coloring must be valid for both Y and Z; $E[X, c] = E[Y, c] \cap E[Z, c]$.

2.3.2. Maximising Constraint Satisfaction Problem (CSP) in linear time.

The goal of the CSP problem is when given a set of variables and a set of constraints on those variables, to find a combination of values for the variables that satisfies all of the constraints. Given a CSP instance whose primal graph is a tree, the number of constraints satisfied can be maximised [2].

Assume the primal graph P is rooted at R. Let $E[v, a]$ denote the maximum number of constraints in T that can be satisfied if $v = a$ where $v =$ variable and $a =$ assignment of variable to an element in the domain. Let the children of v be $v_1, v_2 \dots v_n$, and let $C(a, b)$ be the constraint linking v and $v\{x\}$ when $v = a$ and $v\{x\} = b$ where $x = 1, 2 \dots n$ Thus, we can compute $E[v, a]$ as follows:

$$E[v, a] = \sum \max\{C(a, b) + E[v\{x\}, b] | b \in domain\}$$

This equation simply tracks the optimal number of constraints that can be satisfied by a choice of assignment. Using a dynamic programming approach, the maximum number of constraints that can be satisfied is the max over all $E[R, a]$ where a ranges over all possible assignments [2]

3. Related Work

Treewidth approximation and decomposition has been a field with alot of interest and various algorithms have been developed and enhancements are still being developed. A primary reason for the interest in these algorithms is a consequence of Courcelle's theorem (Theorem 2.3.1). This is a table from the paper [0] that provides an overview of the algorithms so far.

Reference	Appx. $\alpha(k)$	$f(k)$	$g(n)$
Arnborg, Corneil, and Proskurowski [ACP87]	exact	$\mathcal{O}(1)$	n^{k+2}
Robertson and Seymour [RS95]	$4k + 3$	$\mathcal{O}(3^{3k})$	n^2
Matoušek and Thomas [MT91]	$6k + 5$	$k^{\mathcal{O}(k)}$	$n \log^2 n$
Lagergren [Lag96]	$8k + 7$	$k^{\mathcal{O}(k)}$	$n \log^2 n$
Reed [Ree92]	$8k + \mathcal{O}(1)$	$k^{\mathcal{O}(k)}$	$n \log n$
Bodlaender [Bod96]	exact	$2^{\mathcal{O}(k^3)}$	n
Amir [Ami10]	$4.5k$	$\mathcal{O}(2^{3k} k^{3/2})$	n^2
Amir [Ami10]	$(3 + 2/3)k$	$\mathcal{O}(2^{3.6982k} k^3)$	n^2
Amir [Ami10]	$\mathcal{O}(k \log k)$	$\mathcal{O}(k \log k)$	n^4
Feige, Hajiaghayi, and Lee [FHL08]	$\mathcal{O}(k\sqrt{\log k})$	$\mathcal{O}(1)$	$n^{\mathcal{O}(1)}$
Fomin, Todinca, and Villanger [FTV15]	exact	$\mathcal{O}(1)$	1.7347^n
Fomin et al. [FLS+18]	$\mathcal{O}(k^2)$	$\mathcal{O}(k^7)$	$n \log n$
Bodlaender et al. [BDD+16]	$3k + 4$	$2^{\mathcal{O}(k)}$	$n \log n$
Bodlaender et al. [BDD+16]	$5k + 4$	$2^{\mathcal{O}(k)}$	n
Korhonen [Kor21]	$2k + 1$	$2^{\mathcal{O}(k)}$	n
Belbasi and Fürer [BF22]	$5k + 4$	$2^{7.61k}$	$n \log n$
Belbasi and Fürer [BF21]	$5k + 4$	$2^{6.755k}$	$n \log n$
This paper	exact	$2^{\mathcal{O}(k^2)}$	n^4
This paper	$(1 + \varepsilon)k$	$k^{\mathcal{O}(k/\varepsilon)}$	n^4

Figure 1. Each algorithm either outputs a tree decomposition of width atmost αk or returns that the treewidth of the graph is greater than k. The total runtime of each algorithm is $f(k)\dot{g}(n)$ [0]

We will be discussing 3 of these algorithms in the following sections.

3.1. (< 2)-approximation Treewidth algorithm

The *Improved Parameterized Algorithm for Treewidth* [0] paper by Tuukka Korhonen and Daniel Lokshtanov presents a treewidth approximation algorithm with ratio less than 2. It takes as input an n-vertex graph G, an integer k and a rational value $\epsilon \in (0, 1)$. Then, in time $k^{O(k/\epsilon)} \cdot n^{O(1)}$ computes a $(1+\epsilon) \cdot k$ tree decomposition of G or determines that G's treewidth is larger than k. This is the first treewidth approximation algorithm with approximation ratio less than 2. The algorithm works in polynomial space.

The tree decomposition algorithm runs in polynomial space and an interesting feature of the algorithm is that it is not based on a dynamic programming approach. Most work in the domain has been based on dynamic programming. This paper provides two algorithms - an exact treewidth algorithm that runs in time $2^{O(k^2)} \cdot n^4$ and an approximation algorithm that runs in time $k^{O(k/\epsilon)} \cdot n^{O(1)}$.

Theorem 3.1.1. *There is an algorithm that takes as input an n-vertex graph G and an integer k, and in time* $2^{O(k^2)} \cdot n^4$ *either outputs a tree decomposition of G of width at most k or concludes that the treewidth of G is larger than k. Moreover, the algorithm works in space polynomial in n*

Theorem 3.1.2. *There is an algorithm that takes as input an n-vertex graph G, an integer k, and a rational* $\epsilon \in (0, 1)$ *and in time* $k^{O(k/\epsilon)} \cdot n^{O(1)}$ *either outputs a tree decomposition of G of width at most* $(1 + \epsilon) \cdot k$ *or concludes that the treewidth of G is larger than k. Moreover, the algorithm works in space polynomial in* n*.*

The main method used to prove both theorems is based on the "local improvement" method introduced in [cite]. In each step, a decomposition of treewidth greater than k is given and the goal is to determine a decomposition of the tree with width less than or equal to k or to conclude that the treewidth of G is more than k.

SUBSET TREEWIDTH: Given as input a graph G and a set W of vertices either (1) conclude that the treewidth of G is at least $W - 1$, or (2) find a tree decomposition such that W is contained in the union of the non-leaf bags of and all non-leaf bags have size at most $|W| - 1$.

The main method employed by this algorithm is the reduction of the local improvement to SUBSET TREEWIDTH. Let's say the algorithm is given a graph G and an integer k. As established above, the task is to either return that the treewidth of G is more than k or to (1) find a decomposition of width at most k (for the exact algorithm) (2) find a decomposition of width at most $(1+\epsilon) \cdot k$. Along with G, an approximate decomposition of G, (T,bag) is also passed in as input. This decomposition can be computed using any of the other 3,4 or 5-approximation algorithms. If the approximation is not good enough, it is because the largest bag, W in T, is too big. (T,bag) can be made a better decomposition by removing this bag . A new decomposition would then have width at most $|W| - 1$ and fewer bags of size W. This problem of finding a better decomposition can be reduced to SUBSET TREEWIDTH.

torso$_G(X)$: the graph obtained from $G[X]$ by making

$N_G(C)$ a clique for every connected component C of $G \setminus X$

torso tree decomposition: the tuple (X,(T, bag)), where $X \subseteq V(G)$ and (T, bag) is a tree decomposition of $\text{torso}_G(X)$.

torso tree decomposition width: width of (X,(T, bag)) is the width of (T,bag)

Theorem 3.1.3. *If (T,bag) is a tree decomposition of G and W is the largest bag of (T, bag). If there exists a torso tree decomposition (X,(T_0, bag$_0$)) that (1) covers W and (2) has width at most $|W|-2$, then there exists a tree decomposition of G of width at most $|W|-1$ with strictly fewer bags of size W. This new decomposition can be computed in polynomial time*

SUBSET TREEWIDTH directly reduces to the treewidth problem. The paper proves Theorem 3.1.3. The reduction can be done by starting with a decomposition (T,bag) of width $O(k)$. This can be computed using any of the other treewidth approximation algorithms. Then, SUBSET TREEWIDTH is called on the largest bag W of this decomposition. If we can obtain a torso tree decomposition $(X, (T_0, \text{bag}_0))$ that covers W and has width at most $|W| - 2$, then Theorem 3.1.3 yields a decomposition (T_1, bag_1) of size $|W|-1$. Now the process is repeated with (T_1, bag_1) as the new (T, bag). After at most $O(kn)$ iterations, we obtain a tree decomposition with the appropriate width or we find out that the treewidth of G is more than k.

3.2. 5-approximation Treewidth algorithm

The *An Improvement of Reed's Treewidth Algorithm* [6] paper by Mahdi Belbasi and Martin Furer is a very recent paper that offers a faster implementation of Reed's classical treewidth approximation algorithm [cite]. Reed's algorithm has a runtime of $O(2^{k \log k} \cdot n \log n)$ for a 5-approximation. The new paper's algorithm has a smaller dependence on k and runs in $O(2^k \cdot n \log n)$ time. We will first give an overview of Reed's algorithm and then explain the optimizations proposed by this paper.

Reed's algorithm works by splitting a graph into subgraphs and then recursively finding tree decompositions. One of the important tasks for Reed's algorithm is to find a "balanced" separator S that splits the graph $G - S$ into two subgraphs with each graph containing $X, Y \subseteq V$. The algorithm then recursively finds tree decompositions for the subgraphs induced by $X \cup S$ and $Y \cup S$.

To find the balanced separator, the algorithm is applied on groups of vertices and works with their assigned representatives. Then the algorithm branches on the representatives.

Representative vertex: The deepest vertex v in a graph that has at least $n/24k$ vertices. This is found using DFS.

Weight of vertex v: The size of the subtree rooted at v. This is denoted by $w(v)$.

The main idea is if a representative v with a large weight goes to X or Y, then most of vertices in the subtree rooted by v would go to the same set. This is because if a descendant goes to the other side, then at least one vertex in the path connecting the vertex to the representative would be in the separator S. But since S cannot have more than k vertices. So at most $k \cdot \frac{n}{24k} = \frac{n}{24}$ vertices will be in the wrong set. This enables Reed's algorithm to work with a much smaller subset (the subset of representatives) instead of the entire vertex set of the graph. Another thing to note is that checking all the possibilities of representatives going to X,Y or S does not work because if a representative goes to the separator, the vertices in its subtree can go anywhere and thus cannot be constrained. Reed's algorithm handles this by choosing the vertices belonging to the separator in the first step. So if a representative gets placed in the separator, DFS can be done again to find a new group of representatives. After a representative vertex is placed in the separator set, (1) $k = k - 1$ and (2) the other representatives change after DFS procedure.

Assume $W \subseteq V$ of size $3k - 2$. A separator S is found of size at most k to split $G - S$ into two parts L and R. The sizes of $|W \cap X|$ and $|W \cap Y|$ is at most the size of $\frac{2}{3}|W|$. A tree node with $|W \cup S|$ is formed and the recursive calls continue to find the tree decompositions. Reed's algorithm reduces the dependency of n in its runtime from n^2 to $n \log n$ by interspersing the balanced partitions of W with balanced partitions of V. It works best when both V and W are split simultaneously in a balanced way. If only W is partitioned without partitioning V along with it, the graph might get split very unevenly and during the next splitting phase of V, W might not be split at all.

With the alternating splitting procedure, when W is split, the size of W can be at most $6k$. On each side, there are at most $\frac{2}{3}\dot{6}k = 4k$ elements. Splitting by both W and V adds k to the new W, leading to a 7-approximation algorithm. Splitting V is an expensive procedure. So, splitting V only after $\log_{\frac{3}{2}} k$ steps, maintains the runtime and the algorithm also becomes a 5-approximation.

3.2.1. Improvements to Reed's algorithm. The paper reduces the k dependency of the runtime significantly to make the algorithm more practical. There are two main optimizations proposed [6]:

- Larger cutoff compared to Reed's which was $n/24k$
- Avoid branching even if there is a representative vertex going into the separator S.

We will be focusing on the second optimization which is the main improvement proposed by the paper. This allows the representative vertices to go to either the left set, right set or into the separator. When a representative v is added to the separator set, its weight changes to 0. v is also removed from G and all the vertices in the subtree are unmarked. Then, DFS finds the set of representatives from the graph and the 5-approximation treewidth is found recursively.

3.3. Polynomial-time $O(\log OPT)$ Approximation to Treewidth

The *Approximation Algorithms for Treewidth* paper [4] by Eyal Amir presents 4 approximation algorithms for finding optimal tree decompositions. It is the first paper to provide a polynomial-time algorithm that finds an $O(\log OPT)$ width tree decomposition where OPT is the minimal treewidth of the graph. The main idea of this paper's algorithm is that a recursive decomposition of an input graph can lead to an approximation of the optimal tree decomposition. The paper builds on this idea by observing that every graph of treewidth k has a balanced vertex cut of size at most $k+1$. The recursive step then implements an algorithm to find a balanced vertex cut, making sure the level before the recursion is split up in some way as well.

A key difference for the algorithm implemented in the paper is that it only finds a balanced vertex cut of the previous levels before the recursion and can ignore the rest of the graph's balance. The paper then defines a polynomial-time algorithm for finding a balanced vertex cut of a set of vertices W that is within $O(\log W)$ of the optimal cut. Using this and a recursive step, a $O(\log k)$ approximation to the optimal tree decomposition can be found.

The vertex cut observation along with maximum $s-t$ flow algorithms can be used to implement faster algorithms for previously known constant-factor approximations for treewidth. The main task for these problems is setting up the flow problems correctly such that they produce balanced vertex cut solutions. The gain in performance is due to the flow problem set up and because after the recursion, the rest of G is ignored.

We first define two subroutines needed to implement the $O(\log OPT)$ algorithm

(1) Balanced Node Cuts with two weight functions. The first function π_1 as defined in the paper measures the contribution of a vertex separator node to the cost of the cut. The second function π_2 defines the contribution of a separated set node to the weight of the cut. The complete algorithm is shown below. This figure is taken from the original paper [cite] and the function is called *bal-node-cut*

(2) 3-way $\frac{2}{3}$-vertex-separator of $W \subset V$. This is the other main subroutine. Using the bal-node-cut above produces a $\frac{2}{3}$-vertex-separator of W ⊂ V in G(V, E) of size within factor $O(\log |W|)$ from optimal. It does so using procedure bal-node-cut defined above. The weight and cost

```
PROCEDURE bal-node-cut($G$, $\pi_1$, $\pi_2$, $b$, $b'$)
$G = (V, E)$ an undirected graph, $\pi_1 \geq 0$ a cost function for nodes in $V$, $\pi_2 \geq 0$ a
weight function on nodes in $V$, $b \leq 1/2$, $b' < b$ and $b' \leq 1/3$.
    1. Let $G^*(V^*, E^*)$ be a directed graph with edge costs and node weights built
       from $G$ as follows[a]:
        (a) Set $V^* = \{v', v'' | v \in V\}$.
        (b) Set $E^* = \{(v', v'') \mid v \in V\} \cup \{(v'', u') \mid (v, u) \in E\}$.
        (c) Set edge costs in $E^*$: $C(v', v'') = \pi_1(v)$, and $C(v'', u') = \infty$ for $u \neq v$.
        (d) Set node weights in $V^*$: $w(v') = w(v'') = \frac{1}{2}\pi_2(v)$.
    2. Find a $b'$-balanced edge cut $(S, \bar{S})$ in $G^*$ that is within $O(\log p)$ from the
       optimal $b$-balanced edge cut[b], for $p = |\{v \in V \mid \pi_2(v) > 0\}|$.
    3. Return the node cut $(A, C, B)$ for $C = \{v \in V \mid v' \in S,\ v'' \in \bar{S}$ or $v'' \in
       S,\ v' \in \bar{S}\}$, $A = \{v \in V \mid v' \in S$ or $v'' \in S\} \setminus C$, and $B = V \setminus (A \cup C)$.
```

[a]This step is a variant of a well-known translation of a node cut problem to an edge cut problem in a directed graph.

[b]This can be done using any algorithm for balanced edge cut in directed weighted graphs, e.g., [41].

Figure 2. Balanced Node Cuts subroutine from [4]

functions are set as the following for the function call.

- $\pi_1(v) = 1$ for all v in V
- $\pi_2(v) = 1$ for all v in W
- $\pi_2(v) = 0$ for all v in set V-W

This is done to ensure that the cut ignores the weights of the nodes in set $V - W$, but keeps count of all the nodes when computing the cost of the cut regardless of which set they're a member of. The paper calls this subroutine *2-3-vsep-lgk(W ,G)*.

The main $O(\log OPT)$-approximation algorithm uses a *traingulation* approach. It creates cliques and a triangulated graph instead of a tree decomposition. It uses procedure *2-3-vsep-lgk(W ,G)* for its internal computations. The algorithm is called $lgk - triang(G, W, k)$given below.

```
PROCEDURE lgk-triang($G$, $W$, $k$).
$G = (V, E)$ with $|V| = n$, $W \subseteq V$, $k$ integer.
    1. If $n \leq \beta \cdot k \cdot \log k$, then make a clique of $G$. Return.
    2. If $|W| < 2$, then add vertices to $W$ from $G$ such that $|W| = 2$.
    3. Find $X$, an approximate minimum 3-way $\frac{2}{3}$-vertex-separator of $W$ in $G$,
       with $S_1, S_2, S_3$ the three parts separated by $X$ (with $S_1, S_2 \neq \emptyset$, but possibly
       $S_3 = \emptyset$). If $|X| > \beta \cdot k \cdot \log |W|$, then output "the treewidth exceeds $k$" and
       exit.
    4. For $i \leftarrow 1$ to 3 do
        (a) $W_i \leftarrow S_i \cap W$.
        (b) call lgk-triang($G[S_i \cup X]$, $W_i \cup X$, $k$).
    5. Add edges between vertices of $W \cup X$, making a clique of $G[W \cup X]$.
```

Figure 3. The $O(\log OPT)$-approximation algorithm from paper [4]

Theorem 3.3.1. *Let $G(V, E)$ be a graph with n vertices, $k \geq \log n$ an integer and $W \subseteq V$ such that $|W| \leq \gamma \cdot k \cdot \log k$ and $\gamma = c\beta$ for $c = 12 + 3\log k\beta$ a constant. Then, $lgk - triang(G, W, k)$ either outputs correctly that the treewidth of G is more than $k - 1$ or it triangulates G such that the vertices of W form a clique and the clique number of the resulting graph is at most $\frac{4}{3}\gamma k \log k$*

The paper also proves that Theorem 3.1.1 implies that approximation output from the procedure depends inversely on the treewidth of the graph. So, the larger the treewidth of the original graph, the better the approximation we get.

4. Our Algorithm Implementation

In addition to surveying the body of work on treewidth computation, our goal was to implement one of the algorithms that it comprises. We decided on Bodlaender's linear-time algorithm for tree decompositions of small treewidth [7]. The code can be found here. We will summarize its methods here.

4.1. Algorithm description

Given a graph $G = (V, E)$ and arbitrary integer k, Bodlaender's algorithm determines if the graph has treewidth at most k, and if so, computes a tree decomposition with width at most k, using memory and time linear to the number of indices. It relies on the following terminology:

4.1.1. Preliminaries.

- A vertex is *low-degree* if its degree is at most d, and *high-degree* if its degree is greater than d, for some integer d to be determined later.
- A vertex is *friendly* if it is low-degree and at least one of its neighbors is low-degree.
- A vertex is *simplicial* if its neighbors form a clique.
- An *improvement* of a graph G is constructed by adding edges between vertices if they share at least $k+1$ neighbors with degree at most k.
- A vertex is *I-simplicial* if it has degree at most k in G, in addition to being simplicial in improved graph G'.

The intuition for this algorithm relies on the following observations, reproduced here without proof:

Theorem 4.1: For a graph with treewidth at most k, $|E| \leq k|V| - \frac{1}{2}k(k+1)$.

Theorem 4.2: For a graph with treewidth at most k, at least one of the following conditions is true:

(i) The ratio of the number of friendly vertices to total vertices is at least $\frac{1}{4k^2+12k+16}$.
(ii) The ratio of the number of I-simplicial vertices to total vertices is at least $\frac{1}{8k^2+24k+32}$.

4.1.2. Main algorithm. The algorithm proceeds by recursively computing minors of G until one is small enough to be decomposed with width $\leq k$ in linear time using existing methods, and then using it to build the tree decomposition for G. In the process, it rules out insufficiently "narrow" tree decompositions using the two theorems above.

Before recursion, the algorithm answers in the negative if Theorem 4.1 is not satisfied. Then, at each recursive step, it evaluates the following conditions:

Case 1: The graph has at most n vertices for some predetermined n.

In this case, we may use any one of a number of existing k-tree decompositions for small graphs to either find this decomposition or show that it is not possible.

Case 2a: The graph has more than n vertices, and does not satisfy condition (ii) of Theorem 4.2. Then the algorithm proceeds as follows:

- Compute the improved graph of G.
- Compute the I-simplicial vertices of G. If one exists with degree at least $k+1$, it implies that the vertex is connected to a clique of $k+2$ vertices in the improved graph. The existence of a clique lower-bounds the graph's treewidth in relation to the size of the clique. The improved graph has treewidth greater than k, so G does as well.
- If the graph does not satisfy condition (i) of Theorem 4.1, answer in the negative.
- Compute G' by removing all I-simplicial vertices and their adjacent edges from G.
- Recurse over G'.
- If G' has treewidth greater than k, then so does G, and we answer in the negative (this stems from the fact that a minor of a graph is at most as wide as it).
- Otherwise, recursion has found a tree decomposition (B, T) of G' with at most k. From this we construct another such decomposition of G as follows:
- For each I-simplicial vertex v, there exists some vertex $i \in V(T)$ such that $N_G(v) \subseteq B_i$, where $N_G(v)$ refers to the set of v's neighbors in graph G.
- Add to T a new vertex j, adjacent only to i.
- Add to B a new bag $B_j = \{v\} \cup N_G(v)$.

Case 2b: The graph has more than n vertices, and does satisfy condition (ii) of Theorem 4.2. Then the algorithm proceeds as follows:

- Compute a maximal matching $M \subseteq E$. A maximal matching is a maximum-cardinality subset of edges such that no vertex is incident to more than one edge in the subset.
- Compute G' by contracting every edge in M. An edge $e = (u, v)$ is contracted by replacing it with a single vertex uv with $N_G(uv) = N_G(u) \cup N_G(v)$.
- Recurse over G'.
- If G' has treewidth greater than k, then so does G, and we answer in the negative.
- Otherwise, recursion has found a tree decomposition (B, T) of G' with at most k. From this we construct another such decomposition of G as follows:
- Define $G = (V, E)$, $G' = (V', E')$, and M as above. Define $f_M : V \rightarrow V'$ as $f_M(v) = v$ if v is not incident to any edges in M, and otherwise $f_M(v) = w$ where w is the vertex that the edge $(u, v) \in M$ contracted to.

- Compute a tree decomposition (X, T) of G, where $X_i = \{v \in V | f_M(v) \in B_i\}$ and T is as before. This decomposition has width at most $2k + 1$.
- Using an existing linear-time algorithm for treewidth of treewidth-bounded graphs, convert (X, T) to a tree decomposition of G with width at most k if possible, or report that none exists.

4.2. Our implementation

4.2.1. Black-box algorithms. This treewidth computation relies on a number of black-box algorithms, the implementation of which was left open by the authors. We will describe our choices in implementation briefly:

Tree decomposition for small graphs. We arbitrarily set $n = 15$ as the maximum number of vertices for a graph to be considered small and finite. Then we use a triangulation-based algorithm which, given graph G and heuristic X, either computes a tree decomposition with width at most k or reports that none exist.

The triangulation method works as follows: First, it produces an elimination set consisting of vertices and associated bags of vertices:

- Compute an arbitrary vertex ordering $\rho : V \to \mathbb{N}$.
- Define $G_\triangle$ = G.
- For $i = 1, 2, \ldots, |V|$:
- Find the vertex $v \in V(G_\triangle)$ that minimizes heuristic X, breaking ties by lowest $\rho(v)$.
- Remove v from $G_\triangle$.
- Connect every neighbor of v in $G_\triangle$ to every other neighbor.
- Add $(v, N_{G_\triangle}(v))$ to the elimination set.

Then the actual decomposition is produced from the elimination set:

- Initialize the decomposition (B, T) where T consists of a single vertex v_0 corresponding to a single bag $B_0 = V$.
- for each item in the elimination set (v, Z):
- Attempt to find an i such that such that $Z \subseteq B_i$. If none is found, define $i = 0$.
- Add a new vertex corresponding to new bag $Z \cup \{v\}$ and connect it to v_i.

This algorithm runs in $O(|V|^2)$ time, which makes it suitable for small graphs but not graphs of the size Bodlaender's algorithm is meant to handle. All that remains is to decide on the heuristic X. The following two enjoy wide use in both treewidth applications and other graph algorithms:

a.) The *degree* of the vertex, and
b.) The *fill-in* of the vertex, or the number of edges that must be added between its neighbors in order to form a clique.

In practice both heuristics are highly effective and neither is unconditionally superior to the other [3], so we return the better of the two results, minimized over 20 trials with randomized ρ.

Treewidth of treewidth-bounded graphs. This refers to the algorithm used at the end of the recursive step for case 2b. Given a graph G with tree decomposition of width at most l, it either computes a tree decomposition of width at most k or reports that none exist, for any $l > k$. One such algorithm accomplishes this in time singly exponential in k and l [0]. The algorithm is far too complex to be discussed in depth here, but in short, it computes the *trunk* of a partial tree decomposition, a subtree formed by removing non-maximal nodes in the decomposition. In addition, it computes a full set of characteristics for each vertex in a tree decomposition, depending on whether they are start nodes, forget nodes, introduce nodes, or join nodes (see 2.3.1 for the definitions of these terms). This full set consists of representations of every partial path-decomposition rooted at that node.

Definition: A *path decomposition* of graph $G = (V, E)$ is a set of subsets of vertices $X_1, X_2, \ldots, X_r$ subject to the following conditions:

- $V = \bigcup_{i=1}^{r} X_i$.
- For each edge $(u, v) \in E$, there exists some X_i such that $\{u, v\} \in X_i$.
- For $1 \leq i < j < k \leq r$, $X_i \cup X_k \subseteq X_j$.

4.2.2. Other implementation details. We implemented all algorithms in Python 3, and used the python library networkX, for the treewidth heuristic computation for small graphs, among our own classes for the construction, manipulation, and analysis of graphs. The correctness of the algorithm was verified on graphs of up to 200 vertices; anything larger became prohibitively slow to evaluate with the small-graph decomposition algorithms.

5. Conclusion

As we have made apparent, the breadth of applications for bounding treewidth spans a large portion of the NP-hard problem space. Meanwhile, significant improvements to the efficiency of parameterized tree decomposition algorithms continue to be made; for example, Korhonen and Lokshtanov, 2022 resolved a long-standing open problem of whether there existed an $2^{o(k^3)} n^{O(1)}$ time algorithm for exact parameterized treewidth computation [0]. However, many unanswered questions remain regarding treewidth and its application; for example, does there exist a polynomial-time algorithm to compute treewidth for planar graphs? We eagerly anticipate the advancement of our understanding of one of the most significant and fascinating computational bottlenecks within the field of computer science.

Acknowledgments

The authors would like to thank Professor Jelani Nelson and Vishal Raman for facilitating our learning journey throughout this course.

References

[1]

[2] [Online]. Available: http://www.cs.cmu.edu/~odonnell/toolkit13/lecture17.pdf

[3] [Online]. Available: https://scale.iti.kit.edu/_media/resources/theses/ba_wunderlich.pdf

[4] E. Amir, "Approximation algorithms for treewidth," *Algorithmica*, vol. 56, no. 4, p. 448–479, apr 2010.

[5] S. Arnborg, J. Lagergren, and D. Seese, "Easy problems for tree-decomposable graphs," *J. Algorithms*, vol. 12, no. 2, p. 308–340, apr 1991. [Online]. Available: https://doi.org/10.1016/0196-6774(91)90006-K

[6] M. Belbasi and M. Fürer, *An Improvement of Reed's Treewidth Approximation*, 02 2021, pp. 166–181.

[7] H. L. Bodlaender, "A linear time algorithm for finding tree-decompositions of small treewidth," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 226–234. [Online]. Available: https://doi.org/10.1145/167088.167161

[8] ——, "A tourist guide through treewidth," *Acta Cybern.*, vol. 11, no. 1-2, pp. 1–21, 1993. [Online]. Available: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3417

[9] ——, "Treewidth: Structure and algorithms," in *Proceedings of the 14th International Conference on Structural Information and Communication Complexity*, ser. SIROCCO'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 11–25.

[10] H. L. Bodlaender, P. G. Drange, M. S. Dregi, F. V. Fomin, D. Lokshtanov, and M. Pilipczuk, "An o(c$^k n)5 - approximationalgorithmfortreewidth,''in$*Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*$, ser.FOCS'13. USA : IEEEComputerSociety, 2013, p.499–508.[Online].Available :$

H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos, "On exact algorithms for treewidth," *ACM Trans. Algorithms*, vol. 9, no. 1, dec 2012. [Online]. Available: https://doi.org/10.1145/2390176.2390188

H. L. Bodlaender and T. Kloks, "Efficient and constructive algorithms for the pathwidth and treewidth of graphs," *Journal of Algorithms*, vol. 21, no. 2, pp. 358–402, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677496900498

T. Korhonen and D. Lokshtanov, "An improved parameterized algorithm for treewidth," Nov 2022. [Online]. Available: https://arxiv.org/abs/2211.07154